

lualatex.dtx
(LuaTeX-specific support)

David Carlisle and Joseph Wright*

2016/03/13

Contents

1 Overview	1
2 Core TeX functionality	2
3 Plain TeX interface	2
4 Lua functionality	2
4.1 Allocators in Lua	2
4.2 Lua access to TeX register numbers	3
4.3 Module utilities	4
4.4 Callback management	4
5 Implementation	5
5.1 Minimum LuaTeX version	5
5.2 Older L ^A TeX/Plain TeX setup	5
5.3 Attributes	7
5.4 Category code tables	7
5.5 Named Lua functions	9
5.6 Custom whatsits	10
5.7 Lua bytecode registers	10
5.8 Lua chunk registers	10
5.9 Lua loader	10
5.10 Lua module preliminaries	11
5.11 Lua module utilities	12
5.12 Accessing register numbers from Lua	13
5.13 Attribute allocation	15
5.14 Custom whatsit allocation	15
5.15 Bytecode register allocation	15
5.16 Lua chunk name allocation	16
5.17 Lua callback management	16

*Significant portions of the code here are adapted/simplified from the packages `luatex` and `luatexbase` written by Heiko Oberdiek, Élie Roux, Manuel Pégourié-Gonnar and Philipp Gesang.

1 Overview

LuaTeX adds a number of engine-specific functions to TeX. Several of these require set up that is best done in the kernel or need related support functions. This file provides *basic* support for LuaTeX at the L^AT_εX kernel level plus as a loadable file which can be used with plain TeX and L^AT_εX.

This file contains code for both TeX (to be stored as part of the format) and Lua (to be loaded at the start of each job). In the Lua code, the kernel uses the namespace `luatexbase`.

The following `\count` registers are used here for register allocation:

```
\e@alloc@attribute@count Attributes (default 258)
\e@alloc@ccodetable@count Category code tables (default 259)
\e@alloc@luafunction@count Lua functions (default 260)
  \e@alloc@whatsit@count User whatsits (default 261)
  \e@alloc@bytecode@count Lua bytecodes (default 262)
  \e@alloc@luachunk@count Lua chunks (default 263)
```

(`\count 256` is used for `\newmarks` allocation and `\count 257` is used for `\newXeTeXintercharclass` with XeTeX, with code defined in `ltfinal.dtx`). With any L^AT_εX kernel from 2015 onward these registers are part of the block in the extended area reserved by the kernel (prior to 2015 the L^AT_εX kernel did not provide any functionality for the extended allocation area).

2 Core TeX functionality

The commands defined here are defined for possible inclusion in a future L^AT_εX format, however also extracted to the file `ltluatex.tex` which may be used with older L^AT_εX formats, and with plain TeX.

<code>\newattribute</code>	<code>\newattribute{<i>attribute</i>}</code>	Defines a named <code>\attribute</code> , indexed from 1 (<i>i.e.</i> <code>\attribute0</code> is never defined). Attributes initially have the marker value <code>-"7FFFFFFF</code> ('unset') set by the engine.
<code>\newcatcodetable</code>	<code>\newcatcodetable{<i>catcodetable</i>}</code>	Defines a named <code>\catcodetable</code> , indexed from 1 (<code>\catcodetable0</code> is never assigned). A new catcode table will be populated with exactly those values assigned by IniTeX (as described in the LuaTeX manual).
<code>\newluafunction</code>	<code>\newluafunction{<i>function</i>}</code>	Defines a named <code>\luafunction</code> , indexed from 1. (Lua indexes tables from 1 so <code>\luafunction0</code> is not available).
<code>\newwhatsit</code>	<code>\newwhatsit{<i>whatsit</i>}</code>	Defines a custom <code>\whatsit</code> , indexed from 1.
<code>\newluabytecode</code>	<code>\newluabytecode{<i>bytecode</i>}</code>	Allocates a number for Lua bytecode register, indexed from 1.
<code>\newluachunkname</code>	<code>newluachunkname{<i>chunkname</i>}</code>	Allocates a number for Lua chunk register, indexed from 1. Also enters the name of the register (without backslash) into the <code>lua.name</code> table to be used in stack traces.

<code>\catcodetable@initex</code>	Predefined category code tables with the obvious assignments. Note that the
<code>\catcodetable@string</code>	<code>latex</code> and <code>atletter</code> tables set the full Unicode range to the codes predefined by
<code>\catcodetable@latex</code>	the kernel.
<code>\catcodetable@attribute</code>	<code>\setattribute{⟨attribute⟩}{⟨value⟩}</code>
<code>\unsetattribute</code>	<code>\unsetattribute{⟨attribute⟩}</code>

Set and unset attributes in a manner analogous to `\setlength`. Note that attributes take a marker value when unset so this operation is distinct from setting the value to zero.

3 Plain T_EX interface

The `luatex` interface may be used with plain T_EX using `\input{luatex}`. This inputs `luatex.tex` which inputs `etex.src` (or `etex.sty` if used with L^AT_EX) if it is not already input, and then defines some internal commands to allow the `luatex` interface to be defined.

The `luatexbase` package interface may also be used in plain T_EX, as before, by inputting the package `\input luatexbase.sty`. The new version of `luatexbase` is based on this `luatex` code but implements a compatibility layer providing the interface of the original package.

4 Lua functionality

4.1 Allocators in Lua

<code>new_attribute</code>	<code>luatexbase.new_attribute(⟨attribute⟩)</code> Returns an allocation number for the <code>⟨attribute⟩</code> , indexed from 1. The attribute will be initialised with the marker value <code>-"7FFFFFFF</code> ('unset'). The attribute allocation sequence is shared with the T _E X code but this function does <i>not</i> define a token using <code>\attributedef</code> . The attribute name is recorded in the <code>attributes</code> table. A metatable is provided so that the table syntax can be used consistently for attributes declared in T _E X or Lua.
<code>new_whatsit</code>	<code>luatexbase.new_whatsit(⟨whatsit⟩)</code> Returns an allocation number for the custom <code>⟨whatsit⟩</code> , indexed from 1.
<code>new_bytecode</code>	<code>luatexbase.new_bytecode(⟨bytecode⟩)</code> Returns an allocation number for a bytecode register, indexed from 1. The optional <code>⟨name⟩</code> argument is just used for logging.
<code>new_chunkname</code>	<code>luatexbase.new_chunkname(⟨chunkname⟩)</code> Returns an allocation number for a Lua chunk name for use with <code>\directlua</code> and <code>\latelua</code> , indexed from 1. The number is returned and also <code>⟨name⟩</code> argument is added to the <code>lua.name</code> array at that index.

4.2 Lua access to T_EX register numbers

<code>registernumber</code>	<code>luatexbase.registernumber(⟨name⟩)</code> Sometimes (notably in the case of Lua attributes) it is necessary to access a register <i>by number</i> that has been allocated by T _E X. This package provides a function to look up the relevant number using LuaT _E X's internal tables. After for example <code>\newattribute\myattrib</code> , <code>\myattrib</code> would be defined by (say) <code>\myattrib=\attribute15</code> . <code>luatexbase.registernumber("myattrib")</code>
-----------------------------	---

would then return the register number, 15 in this case. If the string passed as argument does not correspond to a token defined by `\attributedef`, `\countdef` or similar commands, the Lua value `false` is returned.

As an example, consider the input:

```

\newcommand\test[1]{%
\typeout{#1: \expandafter\meaning\csname#1\endcsname^^J
\space\space\space\space
\directlua{tex.write(luatexbase.registernumber("#1") or "bad input")}%
}

\test{undefinedrubbish}

\test{space}

\test{hbox}

\test{@MM}

\test{@tempdima}
\test{@tempdimb}

\test{strutbox}

\test{sixt@n}

\attributedef\myattr=12
\myattr=200
\test{myattr}

```

If the demonstration code is processed with LuaL^AT_EX then the following would be produced in the log and terminal output.

```

undefinedrubbish: \relax
    bad input
space: macro:->
    bad input
hbox: \hbox
    bad input
@MM: \mathchar"4E20
    20000
@tempdima: \dimen14
    14
@tempdimb: \dimen15
    15
strutbox: \char"B
    11
sixt@n: \char"10
    16
myattr: \attribute12
    12

```

Notice how undefined commands, or commands unrelated to registers do not produce an error, just return `false` and so print `bad input` here. Note also that

commands defined by `\newbox` work and return the number of the box register even though the actual command holding this number is a `\chardef` defined token (there is no `\boxdef`).

4.3 Module utilities

`provides_module` `luatexbase.provides_module(<info>)`
 This function is used by modules to identify themselves; the `info` should be a table containing information about the module. The required field `name` must contain the name of the module. It is recommended to provide a field `date` in the usual L^AT_EX format `yyyy/mm/dd`. Optional fields `version` (a string) and `description` may be used if present. This information will be recorded in the log. Other fields are ignored.

`module_info` `luatexbase.module_info(<module>, <text>)`

`module_warning` `luatexbase.module_warning(<module>, <text>)`

`module_error` `luatexbase.module_error(<module>, <text>)`

These functions are similar to L^AT_EX's `\PackageError`, `\PackageWarning` and `\PackageInfo` in the way they format the output. No automatic line breaking is done, you may still use `\n` as usual for that, and the name of the package will be prepended to each output line.

Note that `luatexbase.module_error` raises an actual Lua error with `error()`, which currently means a call stack will be dumped. While this may not look pretty, at least it provides useful information for tracking the error down.

4.4 Callback management

`add_to_callback` `luatexbase.add_to_callback(<callback>, <function>, <description>)` Registers the `<function>` into the `<callback>` with a textual `<description>` of the function. Functions are inserted into the callback in the order loaded.

`remove_from_callback` `luatexbase.remove_from_callback(<callback>, <description>)` Removes the callback function with `<description>` from the `<callback>`. The removed function and its description are returned as the results of this function.

`in_callback` `luatexbase.in_callback(<callback>, <description>)` Checks if the `<description>` matches one of the functions added to the list for the `<callback>`, returning a boolean value.

`disable_callback` `luatexbase.disable_callback(<callback>)` Sets the `<callback>` to `false` as described in the Lua_TE_X manual for the underlying `callback.register` built-in. Callbacks will only be set to `false` (and thus be skipped entirely) if there are no functions registered using the callback.

`callback_descriptions` A list of the descriptions of functions registered to the specified callback is returned. `{}` is returned if there are no functions registered.

`create_callback` `luatexbase.create_callback(<name>,metatype,<default>)` Defines a user defined callback. The last argument is a default function or `false`.

`call_callback` `luatexbase.call_callback(<name>,...)` Calls a user defined callback with the supplied arguments.

5 Implementation

¹ `<*2ekernel | tex | latexrelease>`

```
2 (2kernel | latexrelease)\ifx\directlua\@undefined\else
```

5.1 Minimum LuaTeX version

LuaTeX has changed a lot over time. In the kernel support for ancient versions is not provided: trying to build a format with a very old binary therefore gives some information in the log and loading stops. The cut-off selected here relates to the tree-searching behaviour of `require()`: from version 0.60, LuaTeX will correctly find Lua files in the `texmf` tree without ‘help’.

```
3 (latexrelease)\IncludeInRelease{2015/10/01}
4 (latexrelease)          {\newluafunction}{LuaTeX}%
5 \ifnum\luatexversion<60 %
6   \wlog{*****}
7   \wlog{* LuaTeX version too old for ltuatex support *}
8   \wlog{*****}
9   \expandafter\endinput
10 \fi
```

5.2 Older L^AT_EX/Plain T_EX setup

```
11 (*tex)
```

Older L^AT_EX formats don’t have the primitives with ‘native’ names: sort that out. If they already exist this will still be safe.

```
12 \directlua{tex.enableprimitives("",tex.extraprimitives("luatex"))}
13 \ifx\@alloc\@undefined
```

In pre-2014 L^AT_EX, or plain T_EX, load `etex.{sty,src}`.

```
14 \ifx\documentclass\@undefined
15   \ifx\loccount\@undefined
16     \input{etex.src}%
17   \fi
18   \catcode'\@=11 %
19   \outer\expandafter\def\csname newfam\endcsname
20     {\alloc@8\fam\chardef\et@xmaxfam}
21 \else
22   \RequirePackage{etex}
23   \expandafter\def\csname newfam\endcsname
24     {\alloc@8\fam\chardef\et@xmaxfam}
25   \expandafter\let\expandafter\new@mathgroup\csname newfam\endcsname
26 \fi
```

5.2.1 Fixes to `etex.src/etex.sty`

These could and probably should be made directly in an update to `etex.src` which already has some LuaTeX-specific code, but does not define the correct range for LuaTeX.

```
27 % 2015-07-13 higher range in luatex
28 \edef \et@xmaxregs {\ifx\directlua\@undefined 32768\else 65536\fi}
29 % luatex/xetex also allow more math fam
30 \edef \et@xmaxfam {\ifx\Umathchar\@undefined\sixt@n\else@cclvi\fi}
31 \count 270=\et@xmaxregs % locally allocates \count registers
32 \count 271=\et@xmaxregs % ditto for \dimen registers
33 \count 272=\et@xmaxregs % ditto for \skip registers
34 \count 273=\et@xmaxregs % ditto for \muskip registers
```

```

35 \count 274=\et@xmaxregs % ditto for \box registers
36 \count 275=\et@xmaxregs % ditto for \toks registers
37 \count 276=\et@xmaxregs % ditto for \marks classes

```

and 256 or 16 fam. (Done above due to plain/L^AT_EX differences in ltuatex.)

```

38 % \outer\def\newfam{\alloc@8\fam\chardef\et@xmaxfam}

```

End of proposed changes to etex.src

5.2.2 luatex specific settings

Switch to global cf `luatex.sty` to leave room for inserts not really needed for luatex but possibly most compatible with existing use.

```

39 \expandafter\let\csname newcount\expandafter\expandafter\endcsname
40     \csname globcount\endcsname
41 \expandafter\let\csname newdimen\expandafter\expandafter\endcsname
42     \csname globdimen\endcsname
43 \expandafter\let\csname newskip\expandafter\expandafter\endcsname
44     \csname globskip\endcsname
45 \expandafter\let\csname newbox\expandafter\expandafter\endcsname
46     \csname globbox\endcsname

```

Define `\e@alloc` as in latex (the existing macros in `etex.src` hard to extend to further register types as they assume specific 26x and 27x count range. For compatibility the existing register allocation is not changed.

```

47 \chardef\e@alloc@top=65535
48 \let\e@alloc@chardef\chardef
49 \def\e@alloc#1#2#3#4#5#6{%
50   \global\advance#3\@ne
51   \e@ch@ck{#3}{#4}{#5}#1%
52   \allocationnumber#3\relax
53   \global#2#6\allocationnumber
54   \wlog{\string#6=\string#1\the\allocationnumber}}%
55 \gdef\e@ch@ck#1#2#3#4{%
56   \ifnum#1<#2\else
57     \ifnum#1=#2\relax
58       #1\@cclvi
59       \ifx\count#4\advance#1 10 \fi
60     \fi
61     \ifnum#1<#3\relax
62     \else
63       \errmessage{No room for a new \string#4}%
64     \fi
65   \fi}%

```

Two simple L^AT_EX macros used in `ltxltx.sty`.

```

66 \long\def\@gobble#1{}
67 \long\def\@firstofone#1{#1}
68 % Fix up allocations not to clash with |etex.src|.
69 \expandafter\csname newcount\endcsname\e@alloc@attribute@count
70 \expandafter\csname newcount\endcsname\e@alloc@ccodetable@count
71 \expandafter\csname newcount\endcsname\e@alloc@luafunction@count
72 \expandafter\csname newcount\endcsname\e@alloc@whatsit@count

```

```
73 \expandafter\csname newcount\endcsname\@alloc@bytecode@count
74 \expandafter\csname newcount\endcsname\@alloc@luachunk@count
```

End of conditional setup for plain T_EX / old L^AT_EX.

```
75 \fi
76 \</tex>
```

5.3 Attributes

`\newattribute` As is generally the case for the LuaT_EX registers we start here from 1. Notably, some code assumes that `\attribute0` is never used so this is important in this case.

```
77 \ifx\@alloc@attribute@count\@undefined
78 \countdef\@alloc@attribute@count=258
79 \fi
80 \def\newattribute#1{%
81 \e@alloc@attribute@attributedef
82 \e@alloc@attribute@count\m@ne\@alloc@top#1%
83 }
84 \e@alloc@attribute@count=\z@
```

`\setattribute` Handy utilities.

```
\unsetattribute 85 \def\setattribute#1#2{#1=\numexpr#2\relax}
86 \def\unsetattribute#1{#1=-"7FFFFFFF\relax}
```

5.4 Category code tables

`\newcatcodetable` Category code tables are allocated with a limit half of that used by LuaT_EX for everything else. At the end of allocation there needs to be an initialisation step. Table 0 is already taken (it's the global one for current use) so the allocation starts at 1.

```
87 \ifx\@alloc@ccodetable@count\@undefined
88 \countdef\@alloc@ccodetable@count=259
89 \fi
90 \def\newcatcodetable#1{%
91 \e@alloc@catcodetable\chardef
92 \e@alloc@ccodetable@count\m@ne{"8000}#1%
93 \initcatcodetable\allocationnumber
94 }
95 \e@alloc@ccodetable@count=\z@
```

`\catcodetable@initex` Save a small set of standard tables. The Unicode data is read here in using a parser
`\catcodetable@string` simplified from that in `load-unicode-data`: only the nature of letters needs to
`\catcodetable@latex` be detected.

```
\catcodetable@atletter 96 \newcatcodetable\catcodetable@initex
97 \newcatcodetable\catcodetable@string
98 \begingroup
99 \def\setrangeatcode#1#2#3{%
100 \ifnum#1>#2 %
101 \expandafter\@gobble
102 \else
103 \expandafter\@firstofone
```



```

104 \fi
105   {%
106     \catcode#1=#3 %
107     \expandafter\setrange\catcode\expandafter
108     {\number\numexpr#1 + 1\relax}{#2}{#3}
109   }%
110 }
111 \@firstofone{%
112   \catcodetable\catcodetable@initex
113   \catcode0=12 %
114   \catcode13=12 %
115   \catcode37=12 %
116   \setrange\catcode{65}{90}{12}%
117   \setrange\catcode{97}{122}{12}%
118   \catcode92=12 %
119   \catcode127=12 %
120   \savecatcodetable\catcodetable@string
121 \endgroup
122 }%
123 \newcatcodetable\catcodetable@latex
124 \newcatcodetable\catcodetable@atletter
125 \begingroup
126 \def\parseunicodedataI#1;#2;#3;#4\relax{%
127   \parseunicodedataII#1;#3;#2 First>\relax
128 }%
129 \def\parseunicodedataII#1;#2;#3 First>#4\relax{%
130   \ifx\relax#4\relax
131     \expandafter\parseunicodedataIII
132   \else
133     \expandafter\parseunicodedataIV
134   \fi
135   {#1}#2\relax%
136 }%
137 \def\parseunicodedataIII#1#2#3\relax{%
138   \ifnum 0%
139     \if L#21\fi
140     \if M#21\fi
141     >0 %
142     \catcode"#1=11 %
143   \fi
144 }%
145 \def\parseunicodedataIV#1#2#3\relax{%
146   \read\unicoderead to \unicodedataline
147   \if L#2%
148     \count0="#1 %
149     \expandafter\parseunicodedataV\unicodedataline\relax
150   \fi
151 }%
152 \def\parseunicodedataV#1;#2\relax{%
153   \loop
154     \unless\ifnum\count0>#1 %
155       \catcode\count0=11 %
156       \advance\count0 by 1 %
157   \repeat

```

```

158 }%
159 \def\storedpar{\par}%
160 \chardef\unicoderead=\numexpr\count16 + 1\relax
161 \openin\unicoderead=UnicodeData.txt %
162 \loop\unless\ifeof\unicoderead %
163   \read\unicoderead to \unicodedataline
164   \unless\ifx\unicodedataline\storedpar
165     \expandafter\parseunicodedataI\unicodedataline\relax
166   \fi
167 \repeat
168 \closein\unicoderead
169 \@firstofone{%
170   \catcode64=12 %
171   \savecatcodetable\catcodetable@latex
172   \catcode64=11 %
173   \savecatcodetable\catcodetable@atletter
174 }
175 \endgroup

```

5.5 Named Lua functions

`\newluafunction` Much the same story for allocating LuaTeX functions except here they are just numbers so they are allocated in the same way as boxes. Lua indexes from 1 so once again slot 0 is skipped.

```

176 \ifx\e@alloc@luafunction@count\@undefined
177   \countdef\e@alloc@luafunction@count=260
178 \fi
179 \def\newluafunction{%
180   \e@alloc\luafunction\e@alloc\chardef
181   \e@alloc@luafunction@count\m@ne\e@alloc@top
182 }
183 \e@alloc@luafunction@count=\z@

```

5.6 Custom whatsits

`\newwhatsit` These are only settable from Lua but for consistency are definable here.

```

184 \ifx\e@alloc@whatsit@count\@undefined
185   \countdef\e@alloc@whatsit@count=261
186 \fi
187 \def\newwhatsit#1{%
188   \e@alloc\whatsit\e@alloc\chardef
189   \e@alloc@whatsit@count\m@ne\e@alloc@top#1%
190 }
191 \e@alloc@whatsit@count=\z@

```

5.7 Lua bytecode registers

`\newluabytcode` These are only settable from Lua but for consistency are definable here.

```

192 \ifx\e@alloc@bytecode@count\@undefined
193   \countdef\e@alloc@bytecode@count=262
194 \fi
195 \def\newluabytcode#1{%

```

```

196 \e@alloc\luabytecode\e@alloc@chardef
197 \e@alloc@bytecode@count\m@ne\e@alloc@top#1%
198 }
199 \e@alloc@bytecode@count=\z@

```

5.8 Lua chunk registers

`\newluachunkname` As for bytecode registers, but in addition we need to add a string to the `lua.name` table to use in stack tracing. We use the name of the command passed to the allocator, with no backslash.

```

200 \ifx\e@alloc@luachunk@count\@undefined
201 \countdef\e@alloc@luachunk@count=263
202 \fi
203 \def\newluachunkname#1{%
204 \e@alloc\luachunk\e@alloc@chardef
205 \e@alloc@luachunk@count\m@ne\e@alloc@top#1%
206 {\escapechar\m@ne
207 \directlua{lua.name[\the\allocationnumber]="\string#1"}}%
208 }
209 \e@alloc@luachunk@count=\z@

```

5.9 Lua loader

Load the Lua code at the start of every job. For the conversion of T_EX into numbers at the Lua side we need some known registers: for convenience we use a set of systematic names, which means using a group around the Lua loader.

```

210 (2kernel)\everyjob\expandafter{%
211 (2kernel) \the\everyjob
212 \begingroup
213 \attributedef\attributezero=0 %
214 \chardef \charzero =0 %

```

Note name change required on older luatex, for hash table access.

```

215 \countdef \CountZero =0 %
216 \dimendef \dimenzero =0 %
217 \mathchardef \mathcharzero =0 %
218 \muskipdef \muskipzero =0 %
219 \skipdef \skipzero =0 %
220 \toksdef \tokszero =0 %
221 \directlua{require("ltnlua")}
222 \endgroup
223 (2kernel)}
224 (latexrelease)\EndIncludeInRelease

225 % \changes{v1.0b}{2015/10/02}{Fix backing out of \TeX{ } code}
226 % \changes{v1.0c}{2015/10/02}{Allow backing out of Lua code}
227 (latexrelease)\IncludeInRelease{0000/00/00}
228 (latexrelease) \newluafunction{LuaTeX}%
229 (latexrelease)\let\e@alloc@attribute@count\@undefined
230 (latexrelease)\let\newattribute\@undefined
231 (latexrelease)\let\setattribute\@undefined
232 (latexrelease)\let\unsetattribute\@undefined
233 (latexrelease)\let\e@alloc@ccodetable@count\@undefined
234 (latexrelease)\let\newcatcodetable\@undefined

```

```

235 <latexrelease>\let\catcodetable@initex\@undefined
236 <latexrelease>\let\catcodetable@string\@undefined
237 <latexrelease>\let\catcodetable@latex\@undefined
238 <latexrelease>\let\catcodetable@atletter\@undefined
239 <latexrelease>\let\e@alloc@luafunction@count\@undefined
240 <latexrelease>\let\newluafunction\@undefined
241 <latexrelease>\let\e@alloc@luafunction@count\@undefined
242 <latexrelease>\let\newwhatsit\@undefined
243 <latexrelease>\let\e@alloc@whatsit@count\@undefined
244 <latexrelease>\let\newluabytecode\@undefined
245 <latexrelease>\let\e@alloc@bytecode@count\@undefined
246 <latexrelease>\let\newluachunkname\@undefined
247 <latexrelease>\let\e@alloc@luachunk@count\@undefined
248 <latexrelease>\directlua{luatexbase.uninstall()}
249 <latexrelease>\EndIncludeInRelease

250 <2ekernel | latexrelease>\fi
251 </2ekernel | tex | latexrelease>

```

5.10 Lua module preliminaries

```
252 <*lua>
```

Some set up for the Lua module which is needed for all of the Lua functionality added here.

`luatexbase` Set up the table for the returned functions. This is used to expose all of the public functions.

```

253 luatexbase      = luatexbase or { }
254 local luatexbase = luatexbase

```

Some Lua best practice: use local versions of functions where possible.

```

255 local string_gsub      = string.gsub
256 local tex_count       = tex.count
257 local tex_setattribute = tex.setattribute
258 local tex_setcount    = tex.setcount
259 local texio_write_nl  = texio.write_nl

260 local luatexbase_warning
261 local luatexbase_error

```

5.11 Lua module utilities

5.11.1 Module tracking

`modules` To allow tracking of module usage, a structure is provided to store information and to return it.

```
262 local modules = modules or { }
```

`provides_module` Local function to write to the log.

```

263 local function luatexbase_log(text)
264   texio_write_nl("log", text)
265 end

```

Modelled on `\ProvidesPackage`, we store much the same information but with a little more structure.

```

266 local function provides_module(info)
267   if not (info and info.name) then
268     luatexbase_error("Missing module name for provides_module")
269   end
270   local function spaced(text)
271     return text and (" " .. text) or ""
272   end
273   luatexbase_log(
274     "Lua module: " .. info.name
275     .. spaced(info.date)
276     .. spaced(info.version)
277     .. spaced(info.description)
278   )
279   modules[info.name] = info
280 end
281 luatexbase.provides_module = provides_module

```

5.11.2 Module messages

There are various warnings and errors that need to be given. For warnings we can get exactly the same formatting as from `TEX`. For errors we have to make some changes. Here we give the text of the error in the `LATEX` format then force an error from Lua to halt the run. Splitting the message text is done using `\n` which takes the place of `\MessageBreak`.

First an auxiliary for the formatting: this measures up the message leader so we always get the correct indent.

```

282 local function msg_format(mod, msg_type, text)
283   local leader = ""
284   local cont
285   local first_head
286   if mod == "LaTeX" then
287     cont = string_gsub(leader, ".", " ")
288     first_head = leader .. "LaTeX: "
289   else
290     first_head = leader .. "Module " .. msg_type
291     cont = "(" .. mod .. ")"
292     .. string_gsub(first_head, ".", " ")
293     first_head = leader .. "Module " .. mod .. " " .. msg_type .. ":"
294   end
295   if msg_type == "Error" then
296     first_head = "\n" .. first_head
297   end
298   if string.sub(text,-1) ~= "\n" then
299     text = text .. " "
300   end
301   return first_head .. " "
302     .. string_gsub(
303       text
304     .. "on input line "
305       .. tex.inputlineno, "\n", "\n" .. cont .. " "
306     )

```

```

307     .. "\n"
308 end

module\_info Write messages.
module\_warning 309 local function module_info(mod, text)
module\_error 310 texio_write_nl("log", msg_format(mod, "Info", text))
311 end
312 luatexbase.module_info = module_info
313 local function module_warning(mod, text)
314 texio_write_nl("term and log",msg_format(mod, "Warning", text))
315 end
316 luatexbase.module_warning = module_warning
317 local function module_error(mod, text)
318 error(msg_format(mod, "Error", text))
319 end
320 luatexbase.module_error = module_error

```

Dedicated versions for the rest of the code here.

```

321 function luatexbase_warning(text)
322 module_warning("luatexbase", text)
323 end
324 function luatexbase_error(text)
325 module_error("luatexbase", text)
326 end

```

5.12 Accessing register numbers from Lua

Collect up the data from the T_EX level into a Lua table: from version 0.80, LuaT_EX makes that easy.

```

327 local luaregisterbasetable = { }
328 local registermap = {
329   attributezero = "assign_attr"   ,
330   charzero      = "char_given"    ,
331   CountZero     = "assign_int"    ,
332   dimenzero     = "assign_dimen"  ,
333   mathcharzero  = "math_given"    ,
334   muskipzero    = "assign_mu_skip",
335   skipzero      = "assign_skip"   ,
336   tokszero      = "assign_toks"   ,
337 }
338 local createtoken
339 if tex.luatexversion > 81 then
340   createtoken = token.create
341 elseif tex.luatexversion > 79 then
342   createtoken = newtoken.create
343 end
344 local hashtokens = tex.hashtokens()
345 local luatexversion = tex.luatexversion
346 for i,j in pairs (registermap) do
347   if luatexversion < 80 then
348     luaregisterbasetable[hashtokens[i][1]] =
349     hashtokens[i][2]
350   else

```

```

351   luaregisterbasetable[j] = createtoken(i).mode
352   end
353 end

```

`registernumber` Working out the correct return value can be done in two ways. For older LuaTeX releases it has to be extracted from the `hashtokens`. On the other hand, newer LuaTeX's have `newtoken`, and whilst `.mode` isn't currently documented, Hans Hagen pointed to this approach so we should be OK.

```

354 local registernumber
355 if luatexversion < 80 then
356   function registernumber(name)
357     local nt = hashtokens[name]
358     if(nt and luaregisterbasetable[nt[1]]) then
359       return nt[2] - luaregisterbasetable[nt[1]]
360     else
361       return false
362     end
363   end
364 else
365   function registernumber(name)
366     local nt = createtoken(name)
367     if(luaregisterbasetable[nt.cmdname]) then
368       return nt.mode - luaregisterbasetable[nt.cmdname]
369     else
370       return false
371     end
372   end
373 end
374 luatexbase.registernumber = registernumber

```

5.13 Attribute allocation

`new_attribute` As attributes are used for Lua manipulations its useful to be able to assign from this end.

```

375 local attributes=setmetatable(
376 {},
377 {
378   __index = function(t,key)
379     return registernumber(key) or nil
380   end}
381 )
382 luatexbase.attributes=attributes
383 local function new_attribute(name)
384   tex_setcount("global", "e@alloc@attribute@count",
385               tex_count["e@alloc@attribute@count"] + 1)
386   if tex_count["e@alloc@attribute@count"] > 65534 then
387     luatexbase_error("No room for a new \\attribute")
388   end
389   attributes[name]= tex_count["e@alloc@attribute@count"]
390   luatexbase_log("Lua-only attribute " .. name .. " = " ..
391                 tex_count["e@alloc@attribute@count"])
392   return tex_count["e@alloc@attribute@count"]
393 end

```

```
394 luatexbase.new_attribute = new_attribute
```

5.14 Custom whatsit allocation

`new\whatsit` Much the same as for attribute allocation in Lua.

```
395 local function new_whatsit(name)
396   tex_setcount("global", "e@alloc@whatsit@count",
397               tex_count["e@alloc@whatsit@count"] + 1)
398   if tex_count["e@alloc@whatsit@count"] > 65534 then
399     luatexbase_error("No room for a new custom whatsit")
400   end
401   luatexbase_log("Custom whatsit " .. (name or "") .. " = " ..
402               tex_count["e@alloc@whatsit@count"])
403   return tex_count["e@alloc@whatsit@count"]
404 end
405 luatexbase.new_whatsit = new_whatsit
```

5.15 Bytecode register allocation

`new\bytecode` Much the same as for attribute allocation in Lua. The optional $\langle name \rangle$ argument is used in the log if given.

```
406 local function new_bytecode(name)
407   tex_setcount("global", "e@alloc@bytecode@count",
408               tex_count["e@alloc@bytecode@count"] + 1)
409   if tex_count["e@alloc@bytecode@count"] > 65534 then
410     luatexbase_error("No room for a new bytecode register")
411   end
412   luatexbase_log("Lua bytecode " .. (name or "") .. " = " ..
413               tex_count["e@alloc@bytecode@count"])
414   return tex_count["e@alloc@bytecode@count"]
415 end
416 luatexbase.new_bytecode = new_bytecode
```

5.16 Lua chunk name allocation

`new\chunkname` As for bytecode registers but also store the name in the `lua.name` table.

```
417 local function new_chunkname(name)
418   tex_setcount("global", "e@alloc@luachunk@count",
419               tex_count["e@alloc@luachunk@count"] + 1)
420   local chunkname_count = tex_count["e@alloc@luachunk@count"]
421   chunkname_count = chunkname_count + 1
422   if chunkname_count > 65534 then
423     luatexbase_error("No room for a new chunkname")
424   end
425   lua.name[chunkname_count]=name
426   luatexbase_log("Lua chunkname " .. (name or "") .. " = " ..
427               chunkname_count .. "\n")
428   return chunkname_count
429 end
430 luatexbase.new_chunkname = new_chunkname
```


5.17 Lua callback management

The native mechanism for callbacks in LuaTeX allows only one per function. That is extremely restrictive and so a mechanism is needed to add and remove callbacks from the appropriate hooks.

5.17.1 Housekeeping

The main table: keys are callback names, and values are the associated lists of functions. More precisely, the entries in the list are tables holding the actual function as `func` and the identifying description as `description`. Only callbacks with a non-empty list of functions have an entry in this list.

```
431 local callbacklist = callbacklist or { }
```

Numerical codes for callback types, and name-to-value association (the table keys are strings, the values are numbers).

```
432 local list, data, exclusive, simple = 1, 2, 3, 4
433 local types = {
434   list      = list,
435   data      = data,
436   exclusive = exclusive,
437   simple    = simple,
438 }
```

Now, list all predefined callbacks with their current type, based on the LuaTeX manual version 0.80. A full list of the currently-available callbacks can be obtained using

```
\directlua{
  for i,_ in pairs(callback.list()) do
    texio.write_nl("- " .. i)
  end
}
```

in plain LuaTeX. (Some undocumented callbacks are omitted as they are to be removed.)

```
439 local callbacktypes = callbacktypes or {
```

Section 4.1.1: file discovery callbacks.

```
440   find_read_file      = exclusive,
441   find_write_file     = exclusive,
442   find_font_file      = data,
443   find_output_file    = data,
444   find_format_file    = data,
445   find_vf_file        = data,
446   find_map_file       = data,
447   find_enc_file       = data,
448   find_sfd_file       = data,
449   find_pk_file        = data,
450   find_data_file      = data,
451   find_opentype_file  = data,
452   find_truetype_file  = data,
453   find_type1_file     = data,
454   find_image_file     = data,
```

Section 4.1.2: file reading callbacks.

```
455 open_read_file      = exclusive,  
456 read_font_file      = exclusive,  
457 read_vf_file        = exclusive,  
458 read_map_file       = exclusive,  
459 read_enc_file       = exclusive,  
460 read_sfd_file       = exclusive,  
461 read_pk_file        = exclusive,  
462 read_data_file      = exclusive,  
463 read_truetype_file  = exclusive,  
464 read_type1_file     = exclusive,  
465 read_opentype_file  = exclusive,
```

Not currently used by luatex but included for completeness. may be used by a font handler.

```
466 find_cidmap_file    = data,  
467 read_cidmap_file    = exclusive,
```

Section 4.1.3: data processing callbacks.

```
468 process_input_buffer = data,  
469 process_output_buffer = data,  
470 process_jobname      = data,
```

Section 4.1.4: node list processing callbacks.

```
471 contribute_filter    = simple,  
472 buildpage_filter     = simple,  
473 pre_linebreak_filter = list,  
474 linebreak_filter     = list,  
475 append_to_vlist_filter = list,  
476 post_linebreak_filter = list,  
477 hpack_filter         = list,  
478 vpack_filter         = list,  
479 hpack_quality        = list,  
480 vpack_quality        = list,  
481 pre_output_filter    = list,  
482 process_rule         = list,  
483 hyphenate            = simple,  
484 ligaturing           = simple,  
485 kerning              = simple,  
486 insert_local_par     = simple,  
487 mlist_to_hlist      = list,
```

Section 4.1.5: information reporting callbacks.

```
488 pre_dump             = simple,  
489 start_run            = simple,  
490 stop_run             = simple,  
491 start_page_number    = simple,  
492 stop_page_number     = simple,  
493 show_error_hook      = simple,  
494 show_warning_message = simple,  
495 show_error_message   = simple,  
496 show_lua_error_hook  = simple,  
497 start_file           = simple,  
498 stop_file            = simple,
```

Section 4.1.6: PDF-related callbacks.

```

499 finish_pdffile = data,
500 finish_pdfpage = data,
Section 4.1.7: font-related callbacks.

```

```

501 define_font = exclusive,
502 }
503 luatexbase.callbacktypes=callbacktypes

```

`callback.register` Save the original function for registering callbacks and prevent the original being used. The original is saved in a place that remains available so other more sophisticated code can override the approach taken by the kernel if desired.

```

504 local callback_register = callback_register or callback.register
505 function callback.register()
506   luatexbase_error("Attempt to use callback.register() directly\n")
507 end

```

5.17.2 Handlers

The handler function is registered into the callback when the first function is added to this callback's list. Then, when the callback is called, the handler takes care of running all functions in the list. When the last function is removed from the callback's list, the handler is unregistered.

More precisely, the functions below are used to generate a specialized function (closure) for a given callback, which is the actual handler.

Handler for data callbacks.

```

508 local function data_handler(name)
509   return function(data, ...)
510     for _,i in ipairs(callbacklist[name]) do
511       data = i.func(data,...)
512     end
513     return data
514   end
515 end

```

Handler for `exclusive` callbacks. We can assume `callbacklist[name]` is not empty: otherwise, the function wouldn't be registered in the callback any more.

```

516 local function exclusive_handler(name)
517   return function(...)
518     return callbacklist[name][1].func(...)
519   end
520 end

```

Handler for list callbacks.

```

521 local function list_handler(name)
522   return function(head, ...)
523     local ret
524     local alltrue = true
525     for _,i in ipairs(callbacklist[name]) do
526       ret = i.func(head, ...)
527       if ret == false then
528         luatexbase_warning(
529           "Function '" .. i.description .. "' returned false\n"
530           .. "in callback '" .. name .. "'")
531       )

```

```

532         break
533     end
534     if ret ~= true then
535         alltrue = false
536         head = ret
537     end
538 end
539 return alltrue and true or head
540 end
541 end

```

Handler for simple callbacks.

```

542 local function simple_handler(name)
543     return function(...)
544         for _,i in ipairs(callbacklist[name]) do
545             i.func(...)
546         end
547     end
548 end

```

Keep a handlers table for indexed access.

```

549 local handlers = {
550     [data]      = data_handler,
551     [exclusive] = exclusive_handler,
552     [list]      = list_handler,
553     [simple]     = simple_handler,
554 }

```

5.17.3 Public functions for callback management

Defining user callbacks perhaps should be in package code, but impacts on `add_to_callback`. If a default function is not required, it may be declared as `false`. First we need a list of user callbacks.

```

555 local user_callbacks_defaults = { }

```

`create_callback` The allocator itself.

```

556 local function create_callback(name, ctype, default)
557     if not name or name == ""
558     or not ctype or ctype == ""
559     then
560         luatexbase_error("Unable to create callback:\n" ..
561             "valid callback name and type required")
562     end
563     if callbacktypes[name] then
564         luatexbase_error("Unable to create callback '" .. name ..
565             "':\ncallback type disallowed as name")
566     end
567     if default ~= false and type (default) ~= "function" then
568         luatexbase_error("Unable to create callback '" .. name ..
569             ":\ndefault is not a function")
570     end
571     user_callbacks_defaults[name] = default
572     callbacktypes[name] = types[ctype]
573 end
574 luatexbase.create_callback = create_callback

```

`call_callback` Call a user defined callback. First check arguments.

```
575 local function call_callback(name,...)
576   if not name or name == "" then
577     luatexbase_error("Unable to create callback:\n" ..
578                     "valid callback name required")
579   end
580   if user_callbacks_defaults[name] == nil then
581     luatexbase_error("Unable to call callback '" .. name
582                     .. "':\nunknown or empty")
583   end
584   local l = callbacklist[name]
585   local f
586   if not l then
587     f = user_callbacks_defaults[name]
588     if l == false then
589       return nil
590     end
591   else
592     f = handlers[callbacktypes[name]](name)
593   end
594   return f(...)
595 end
596 luatexbase.call_callback=call_callback
```

`add_to_callback` Add a function to a callback. First check arguments.

```
597 local function add_to_callback(name, func, description)
598   if not name or name == "" then
599     luatexbase_error("Unable to register callback:\n" ..
600                     "valid callback name required")
601   end
602   if not callbacktypes[name] or
603     type(func) ~= "function" or
604     not description or
605     description == "" then
606     luatexbase_error(
607       "Unable to register callback.\n\n"
608       .. "Correct usage:\n"
609       .. "add_to_callback(<callback>, <function>, <description>)"
610     )
611   end
```

Then test if this callback is already in use. If not, initialise its list and register the proper handler.

```
612   local l = callbacklist[name]
613   if l == nil then
614     l = { }
615     callbacklist[name] = l
```

If it is not a user defined callback use the primitive callback register.

```
616     if user_callbacks_defaults[name] == nil then
617       callback_register(name, handlers[callbacktypes[name]](name))
618     end
619   end
```

Actually register the function and give an error if more than one `exclusive` one is registered.

```
620 local f = {
621   func      = func,
622   description = description,
623 }
624 local priority = #l + 1
625 if callbacktypes[name] == exclusive then
626   if #l == 1 then
627     luatexbase_error(
628       "Cannot add second callback to exclusive function\n'" ..
629       name .. "'")
630   end
631 end
632 table.insert(l, priority, f)
```

Keep user informed.

```
633 luatexbase_log(
634   "Inserting '" .. description .. "' at position "
635   .. priority .. " in '" .. name .. "'."
636 )
637 end
638 luatexbase.add_to_callback = add_to_callback
```

`remove_from_callback` Remove a function from a callback. First check arguments.

```
639 local function remove_from_callback(name, description)
640   if not name or name == "" then
641     luatexbase_error("Unable to remove function from callback:\n" ..
642       "valid callback name required")
643   end
644   if not callbacktypes[name] or
645     not description or
646     description == "" then
647     luatexbase_error(
648       "Unable to remove function from callback.\n\n"
649       .. "Correct usage:\n"
650       .. "remove_from_callback(<callback>, <description>)"
651     )
652   end
653   local l = callbacklist[name]
654   if not l then
655     luatexbase_error(
656       "No callback list for '" .. name .. "'\n")
657   end
```

Loop over the callback's function list until we find a matching entry. Remove it and check if the list is empty: if so, unregister the callback handler.

```
658 local index = false
659 for i,j in ipairs(l) do
660   if j.description == description then
661     index = i
662     break
663   end
664 end
```

```

665 if not index then
666     luatexbase_error(
667         "No callback '" .. description .. "' registered for '" ..
668         name .. "'\n")
669 end
670 local cb = l[index]
671 table.remove(l, index)
672 luatexbase_log(
673     "Removing '" .. description .. "' from '" .. name .. "'."
674 )
675 if #l == 0 then
676     callbacklist[name] = nil
677     callback_register(name, nil)
678 end
679 return cb.func,cb.description
680 end
681 luatexbase.remove_from_callback = remove_from_callback

```

`in_callback` Look for a function description in a callback.

```

682 local function in_callback(name, description)
683     if not name
684         or name == ""
685         or not callbacklist[name]
686         or not callbacktypes[name]
687         or not description then
688         return false
689     end
690     for _, i in pairs(callbacklist[name]) do
691         if i.description == description then
692             return true
693         end
694     end
695     return false
696 end
697 luatexbase.in_callback = in_callback

```

`disable_callback` As we subvert the engine interface we need to provide a way to access this functionality.

```

698 local function disable_callback(name)
699     if(callbacklist[name] == nil) then
700         callback_register(name, false)
701     else
702         luatexbase_error("Callback list for " .. name .. " not empty")
703     end
704 end
705 luatexbase.disable_callback = disable_callback

```

`callback_descriptions` List the descriptions of functions registered for the given callback.

```

706 local function callback_descriptions (name)
707     local d = {}
708     if not name
709         or name == ""
710         or not callbacklist[name]
711         or not callbacktypes[name]

```

```

712     then
713     return d
714   else
715   for k, i in pairs(callbacklist[name]) do
716     d[k]= i.description
717   end
718   end
719   return d
720 end
721 luatexbase.callback_descriptions =callback_descriptions

```

`uninstall` Unlike at the \TeX level, we have to provide a back-out mechanism here at the same time as the rest of the code. This is not meant for use by anything other than `latexrelease`: as such this is *deliberately* not documented for users!

```

722 local function uninstall()
723   module_info(
724     "luatexbase",
725     "Uninstalling kernel luatexbase code"
726   )
727   callback.register = callback_register
728   luatexbase = nil
729 end
730 luatexbase.uninstall = uninstall

```

```

731  $\langle$ /lua $\rangle$ 

```

Reset the catcode of @.

```

732  $\langle$ tex $\rangle$ \catcode'\@=\etacatcode\relax

```